Sheet 3 and solution

Problem (1)

Given a binary pattern in some memory location, is it possible to tell whether this pattern represents a machine instruction or a number?

Ansewr:

No, any binary pattern can be interpreted as a number or as an instruction

Problem (2)

Write a program that can evaluate the expression

$$A \times B + C \times D$$

in a single-accumulator processor. Assume that the processor has Load, Store, Multiply, and Add instructions, and that all values fit in the accumulator.

Answer:

A program for the expression is:

```
Load      A
Multiply  B
Store     RESULT
Load      C
Multiply  D
Add       RESULT
Store     RESULT
```

Problem (3)

Byte-Sorting Program (Page 87)

Sorting a list of n bytes stored in memory into ascending alphabetical order using the Selection Sort Algorithm.

- C-Language Program

```
for  (j=n-1; j > 0; j=j-1)
{ for (k=j-1; k >= 0; k=k-1)
    { if  (List[k] > List[j])
        {  Temp = List[k];
           List[k] = List[j];
           List[j] = Temp;
        }
    }
}
```

Assembly sorting program

```
        Move            #List, R0       load List into R0
        Move            N, R1           initialize outer loop
        Subtract        #1, R1          R1 (j = N - 1)
Outer   Move            R1, R2          initialize inner loop
        Subtract        #1, R2          R2 (k = j – 1)
        MoveByte        (R0,R1), R3     load List(j) into R3
Inner   CompareByte     R3, (R0,R2)     If List(k) <= [R3]
        Branch <= 0     Next            don’t exchange
        MoeByte         (R0,R2), R4     otherwise, exchange
        MoveByte        R3, (R0,R2)     List(k) with List(j) and
        MoveByte        R4, (R0,R1)     load new maximum into R3
        MoveByte        R4, R3          R4 serves as Temp
Next    Decrement       R2              decrement inner loop counter R2
```

```
Branch >= 0    Inner         if loop not finished
Decrement      R1            decrement outer loop counter R1
Branch >= 0    Outer         if loop not finished
```

Problem (4)

Both of the following statements cause the value 300 to be stored in location 1000, but at different times.

```
ORIGIN      1000
DATAWORD    300
```

and

```
Move   #300,1000
```

Explain the difference.

Answer:

The first two instruction are assembler directives which are executed by the assembler before the execution of the normal instructions of the program such as the second instruction

Move #300,1000.

Problem (5)

Rewrite the assembly program to compute the dot product of two vectors A, B of n-bits using a subroutine.

Answer:

Calling program:

```
Move   #Avec, R1       R1 points to vector A
```

```
        Move    #Bvec, R2          R2 points to vector B
        Move    N, R3              R3 serves as the vector size
        Call Sub
        Move    R0, DotProduct     Store the result into Memory
```

Subroutine:

```
SUB     Clear    R0              R0 accumulates the dot product
Loop    Move     (R1)+, R4       load the first number into R4
        Multiply (R2)+, R4       Computes the product
        Add      R4, R0          Add to previous Sum
        Decrement  R3            Decrement the vector Size
        Branch>0  Loop           Loop again if not done
        Return
```

Problem(6)

Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has an address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is

a) Direct. b) indirect. c) PC relative. d) indexed

Solution:

a) X3=X2
b) X3=(X2)
c) X3=X1+X2+1
d) X3=X2+X4

Problem (7)

11.3 An address field in an instruction contains decimal value 14. Where is the corresponding operand located for:

a) immediate addressing?

b) direct addressing?

c) indirect addressing?

d) register addressing?

e) register indirect addressing?

Solution:

Instruction

Opcode	Address 14

a) 14 (The address field).

b) Memory location 14.

c) The memory location whose address is in memory location 14.

d) Register 14.

e) The memory location whose address is in register 14.

Problem(8) (Problem 2.18)

A FIFO queue of bytes is to be implemented in the memory, occupying a fixed region of k bytes. You need two pointers, an IN pointer and an OUT pointer. The IN pointer keeps track of the location where the next byte is to be appended to the queue, and the OUT pointer keeps track of the location containing the next byte to be removed from the queue.

(*a*) As data items are added to the queue, they are added at successively higher addresses until the end of the memory region is reached. What happens next, when a new item is to be added to the queue?

(*b*) Choose a suitable definition for the IN and OUT pointers, indicating what they point to in the data structure. Use a simple diagram to illustrate your answer.

(*c*) Show that if the state of the queue is described only by the two pointers, the situations when the queue is completely full and completely empty are indistinguishable.

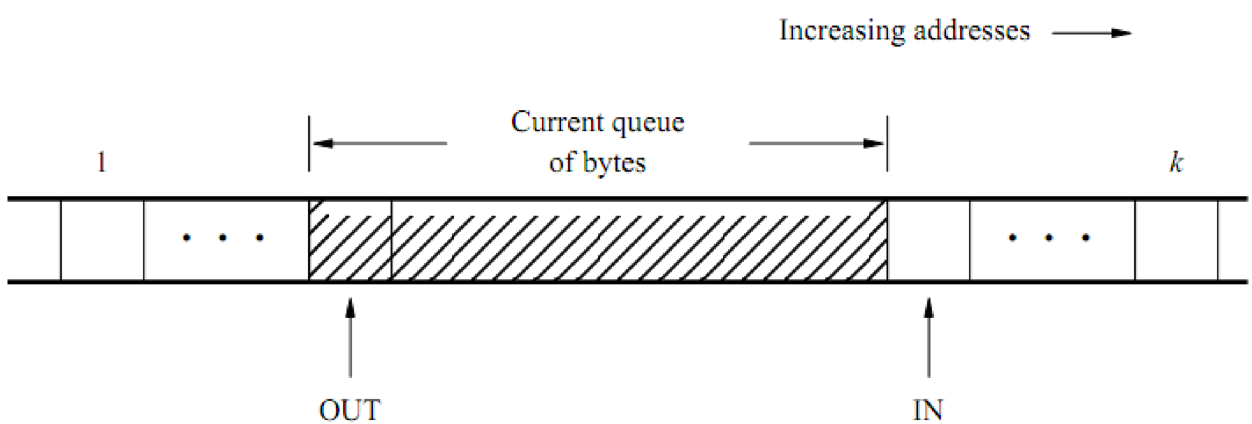
(*d*) What condition would you add to solve the problem in part *c*?

(*e*) Propose a procedure for manipulating the two pointers IN and OUT to append and remove items from the queue.

Solution

(*a*) Wraparound must be used. That is, the next item must be entered at the beginning of the memory region, assuming that location is empty.

(*b*) A current queue of bytes is shown in the memory region from byte location 1 to byte location k in the following diagram.



The IN pointer points to the location where the next byte will be appended to the queue. If the queue is not full with k bytes, this location is empty, as shown in the diagram.

The OUT pointer points to the location containing the next byte to be removed from the queue. If the queue is not empty, this location contains a valid byte, as shown in the diagram.

Initially, the queue is empty and both IN and OUT point to location 1.

(c) Initially, as stated in Part b, when the queue is empty, both the IN and OUT pointers point to location 1. When the queue has been filled with k bytes and none of them have been removed, the OUT pointer still points to location 1. But the IN pointer must also be pointing to location 1, because (following the wraparound rule) it must point to the location where the next byte will be appended. Thus, in both cases, both pointers point to location 1; but in one case the queue is empty, and in the other case it is full.

(d) One way to resolve the problem in Part (c) is to maintain at least one empty location at all times. That is, an item cannot be appended to the queue if ([IN] + 1) Modulo k = [OUT]. If this is done, the queue is empty only when [IN] = [OUT].

(e) Append operation:

- LOC $\leftarrow$ [IN]   Temporary storage for the contents of IN pointer
- IN $\leftarrow$ ([IN] + 1) Modulo k
- If [IN] = [OUT], queue is full. Restore contents of IN to contents of LOC and indicate failed append operation, that is, indicate that the queue was full. Otherwise, store new item at LOC.

Remove operation:

- If [IN] = [OUT], the queue is empty. Indicate failed remove operation, that is, indicate that the queue was empty. Otherwise, read the item pointed to by OUT and perform OUT $\leftarrow$ ([OUT] + 1) Modulo k.

Problem(9) (Problem 2.19)

Consider the queue structure described in Problem 2.18. Write APPEND and REMOVE routines that transfer data between a processor register and the queue. Be careful to inspect and update the state of the queue and the pointers each time an operation is attempted and performed.

Solution

Use the following register assignment:

R0 – Item to be appended to or removed from queue
R1 – IN pointer
R2 – OUT pointer
R3 – Address of beginning of queue area in memory
R4 – Address of end of queue area in memory
R5 – Temporary storage for [IN] during append operation

Assume that the queue is initially empty, with [R1] = [R2] = [R3].

The following APPEND and REMOVE routines implement the procedures required in Part (*e*) of Problem 2.18.

APPEND routine:

```
          Move       R1,R5
          Increment  R1          Increment IN pointer
          Compare    R1,R4         Modulo k.
          Branch≥0   CHECK
          Move       R3,R1
CHECK     Compare    R1,R2       Check if queue is full.
          Branch=0   FULL
          MoveByte   R0,(R5)     If queue not full, append item.
          Branch     CONTINUE
FULL      Move       R5,R1       Restore IN pointer and send
          Call       QUEUEFULL     message that queue is full.
CONTINUE  ...
```

REMOVE routine:

```
              Compare     R1,R2          Check if queue is empty.
              Branch=0    EMPTY          If empty, send message.
              MoveByte    (R2)+,R0       Otherwise, remove byte and
              Compare     R2,R4            increment R2 Modulo k.
              Branch≥0    CONTINUE
              Move        R3,R2
              Branch      CONTINUE
EMPTY         Call        QUEUEEMPTY
CONTINUE      ...
```